FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage

Caroline Lemieux
University of California, Berkeley, USA
clemieux@cs.berkeley.edu

Koushik Sen
University of California, Berkeley, USA
ksen@cs.berkeley.edu

ABSTRACT

In recent years, fuzz testing has proven itself to be one of the most effective techniques for finding correctness bugs and security vulnerabilities in practice. One particular fuzz testing tool, American Fuzzy Lop (AFL), has become popular thanks to its ease-of-use and bug-finding power. However, AFL remains limited in the bugs it can find since it simply does not cover large regions of code. If it does not cover parts of the code, *it will not find bugs there*. We propose a two-pronged approach to increase the coverage achieved by AFL. First, the approach automatically identifies branches exercised by few AFL-produced inputs (*rare* branches), which often guard code that is empirically hard to cover by naïvely mutating inputs. The second part of the approach is a novel *mutation mask* creation algorithm, which allows mutations to be biased towards producing inputs hitting a given rare branch. This mask is dynamically computed during fuzz testing and can be adapted to other testing targets. We implement this approach on top of AFL in a tool named FairFuzz. We conduct evaluation on real-world programs against state-of-the-art versions of AFL. We find that on these programs FairFuzz achieves high branch coverage at a faster rate that state-of-the-art versions of AFL. In addition, on programs with nested conditional structure, it achieves sustained increases in branch coverage after 24 hours (average 10.6% increase). In qualitative analysis, we find that FairFuzz has an increased capacity to automatically discover keywords.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

fuzz testing, coverage-guided greybox fuzzing, rare branches

ACM Reference Format:
Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238176

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238176

1 INTRODUCTION

Fuzz testing has emerged as one of the most effective testing techniques for finding correctness bugs and security vulnerabilities in real-world software systems. It has been used successfully by major software companies such as Microsoft [21] and Google [6, 18, 40] for security testing and quality assurance. The success of *coverage-guided greybox fuzzing* (CGF) in particular has gained attention both in practice and in the research community [10, 11, 37, 42, 46]. One of the leading CGF tools, American Fuzzy Lop (or simply AFL) [51], has found vulnerabilities in a broad array of programs, including Web browsers (e.g. Firefox, Internet Explorer), network tools (e.g., tcpdump, wireshark), image processors (e.g., ImageMagick, libtiff), various system libraries (e.g., OpenSSH, PCRE), C compilers (e.g., GCC, LLVM), and interpreters (for Perl, PHP, JavaScript).

Coverage-guided greybox fuzzing is based on the observation that *increasing program coverage often leads to better crash detection*. The actual fuzzing process starts with a set of user-provided seed inputs. It then mutates the seed inputs with byte-level operations. It runs the program under test on the mutated inputs and collects program coverage information. Finally, it saves the mutated inputs which are *interesting* according to the coverage information—the ones that discover new coverage. It continually repeats the process, but starting with these interesting mutated inputs instead of the user-provided inputs.

While many of the individual test inputs it generates may be garbage, due to its low computational overhead, CGF generates test inputs much faster than more sophisticated methods such as symbolic execution [17, 34] and dynamic symbolic execution (a.k.a. concolic testing) techniques [5, 7, 13, 16, 21, 22, 36, 45, 47]. In practice, this trade-off has paid off, and CGF has found numerous correctness bugs and security vulnerabilities in widely used software [3, 10, 11, 37, 46, 51].

Although the goal of AFL and other CGF tools is to find assertion violations and crashes as quickly as possible, their core search strategies are *based on coverage feedback*—AFL tries to maximize the coverage it achieves. This is because there is no way to find bugs or crashes at a particular program location unless that location is covered by a test input. However, while experimenting with AFL and its extensions, we observed that AFL often fails to cover key program functionalities. For example, AFL did not cover colorspace conversion code in `djpeg`, attribute list processing code in `xmllint`, and a large variety of packet structures in `tcpdump`. Therefore, AFL cannot be expected to find bugs in these functionalities. Put succinctly, if AFL does not cover some program regions, *it will not find bugs in those regions.*

We propose a lightweight technique, called FairFuzz, which helps AFL achieve better coverage. This technique requires *no extra instrumentation* beyond AFL's regular instrumentation, unlike

some other recently proposed fuzzing techniques [15, 37, 42, 46]. Thus, the technique preserves AFL's ease-of-use. While we focus on branch coverage, our proposed technique could easily be modified for other kinds of coverage and testing objectives. FairFuzz is based on a novel mutation strategy that increases the probability of hitting the code locations that have been hit by few of AFL's previously-generated inputs.

FairFuzz works in two main steps. First, it identifies the program branches that are rarely hit by previously-generated inputs. We call such branches *rare branches*. These rare branches guard under-explored functionalities of the program. By generating more random inputs hitting these rare branches, FairFuzz greatly increases the coverage of the parts of the code guarded by them.

Second, FairFuzz uses a novel lightweight mutation technique to increase the probability of hitting these rare branches. The mutation strategy is based on the observation that certain parts of an input already hitting a rare branch are crucial to satisfy the conditions necessary to hit that branch. Therefore, to generate more inputs hitting the rare branch via mutation, the parts of the input that are crucial for hitting the branch should not be mutated. FairFuzz identifies these crucial parts of the input by performing a number of small mutation experiments. Later, in test input generation, it avoids mutating these crucial parts of the input. This mutation strategy is orthogonal to approaches that try to increase crashes found by AFL by helping AFL pass magic byte checks [37, 46] or by customizing power schedules [10, 11] and can be combined with them to simultaneously increase code coverage and the number of bugs or crashes discovered.

We implemented our technique, FairFuzz, on top of AFL. We evaluated FairFuzz against three popular versions of AFL. We compare to AFLFast [11], an extension of AFL which prioritizes inputs that hit rare paths through the program, but does not change the mutation techniques of AFL. We conduct our evaluation on nine real-world benchmarks, including those used to evaluate AFLFast. We repeat our experiments and provide measures of variability when comparing techniques in our evaluation. We find that on these benchmarks FairFuzz achieves high branch coverage at a faster rate that state-of-the-art versions of AFL (average 6.0% increase after 1 hour, and 3.5% increase after 5 hours). Prior work demonstrates that after a certain amount of coverage is achieved, even small increases in branch coverage can yield more bug finding power [7]. FairFuzz has a stronger advantage on programs with nested conditional structure, obtaining sustained increases in branch coverage (average 10.6% increase after 24 hours), and, according to qualitative coverage analysis, better discovering complex keywords.

In summary, we make the following contributions:

- We propose a novel lightweight mutation masking strategy to increase the chance of hitting the program regions that are missed by previously generated inputs. We describe a method to implement mutation masking which smoothly integrates with the usual fuzz testing mutation procedure.
- We develop an open-source[1] implementation of mutation masking targeted to rare branches on top of AFL, named *FairFuzz*.

[1] https://github.com/carolemieux/afl-rb

Algorithm 1 AFL algorithm.

```
1:  procedure FuzzTest(Prog, Seeds)
2:    Queue ← Seeds
3:    while true do                          ▷ begin a queue cycle
4:      for input in Queue do
5:        if ¬isWorthFuzzing(input) then
6:          continue
7:        score ← performanceScore(Prog,input)
8:        for 0 ≤ i < |input| do
9:          for mutation in deterministicMutationTypes do
10:           newinput ←mutate(input, mutation, i)
11:           runAndSave(Prog, newinput, Queue)
12:       for 0 ≤ i < score do
13:         newinput ←mutateHavoc(input)
14:         runAndSave(Prog, newinput, Queue)
15: procedure mutateHavoc(Prog, input)
16:   numMutations ← randomBetween(1,256)
17:   newinput← input
18:   for 0 ≤ i <numMutations do
19:     mutation ← randomMutationType
20:     position ← randomBetween(0, |newinput|)
21:     newinput ← mutate(newinput, mutation, position)
22:   return newinput
23: procedure runAndSave(Prog, input, Queue)
24:   runResults ← run(Prog, input)
25:   if newCoverage(runResults) then
26:     addToQueue(input, Queue)
```

- We perform evaluation of FairFuzz against different state-of-the-art versions of AFL on real-world benchmarks.

We detail the general method and implementation of FairFuzz in Section 3 and the performance results in Section 4. We will begin in Section 2 with a more detailed overview of AFL, its current limitations, and how to overcome these with our method.

2 OVERVIEW

Our proposed technique, FairFuzz, is built on top of American Fuzzy Lop (AFL) [51]. AFL is a popular greybox mutation-based fuzz tester. *Greybox* fuzz testers [3, 51] are designated as such since, unlike *whitebox* fuzz testers [13, 21, 22, 44], they do not do any source code analysis, but, unlike pure *blackbox* fuzz testers [30], they use limited feedback from the program under test to guide their fuzzing strategy. Next we give a brief description of AFL, illustrate one of its limitations, and motivate the need for FairFuzz.

2.1 AFL Overview

To fuzz test programs, AFL generates random inputs. However, instead of generating these inputs from scratch, it selects a set of previously generated inputs and mutates them to derive new inputs. The overall AFL fuzzing algorithm is given in Algorithm 1.

The fuzzing routine takes as input a program and a set of user-provided *seed inputs*. The seed inputs are used to initialize a *queue* (Line 2) of inputs. AFL goes through this queue (Line 4), selects an input to mutate (Line 5), mutates the input (Lines 10, 13), runs the

program on and, simultaneously, collects the coverage information for the mutated inputs (Line 24), and finally adds these mutated inputs to the queue if they achieve new coverage (Line 26). An entire pass through the queue is called a *cycle*. Cycles are repeated (Line 3) until the fuzz testing procedure is stopped by the user.

AFL's mutation strategies assume the input to the program under test is a sequence of bytes, and can be treated as such during mutation. AFL mutates inputs in two main stages: the *deterministic* (Algorithm 1, Lines 8-11) stages and the *havoc* (Lines 12-14) stage. All the deterministic mutation stages operate by traversing the input under mutation and applying a mutation at each position in this input. These mutations (Line 9) include bit flipping, byte flipping, arithmetic increment and decrement of integer values, replacing of bytes with "interesting" integer values (0, `MAX_INT`), etc. The number of mutated inputs produced in each of these stages is governed by the length of the input being mutated (Line 8). On the other hand, the havoc stage works by applying a sequence of random mutations—setting random bytes to random values, deleting or cloning subsequences of the input—to the input being mutated to produce a new input. Several mutations are applied to the original input (Line 21) before running it through the program (Line 14). The number of total havoc-mutated inputs to be produced is determined by a performance score, *score* (Line 7).

2.2 AFL Coverage Calculation

Above, we mentioned the role that coverage information plays in the AFL procedure. The use of this information is one of AFL's key innovations. Specifically, AFL uses this information to select inputs for mutation and save new inputs, saving only those that have achieved new program coverage. In order to collect this coverage information efficiently, AFL inserts instrumentation into the program under test. To track coverage, it first associates each basic block with a random number via instrumentation. The random number is treated as the *unique ID* of the basic block. The basic block IDs are then used to generate unique IDs for the transitions between pairs of basic blocks. In particular, for a transition from basic block A to B, AFL uses the IDs of each basic block—$ID(A)$ and $ID(B)$, respectively—to define the ID of the transition, $ID(A \rightarrow B)$, as follows:

$$ID(A \rightarrow B) \stackrel{def}{=} (ID(A) \gg 1) \oplus ID(B).$$

Right-shifting ($\gg$) the basic block ID of the transition start block (A) ensures that the transition from A to B has a different ID from the transition from B to A. We associate the notion of basic block transition with that of a *branch* in the program's control flow graph, and throughout the paper we will use the term branch to refer to this AFL-defined basic block transition unless stated otherwise.

The coverage of the program under test on a given input is collected as a set of pairs of the form (*branch ID*, *branch hits*). If a (*branch ID*, *branch hits*) pair is present in the coverage set, it denotes that during the execution of the program on the input, the branch with ID *branch ID* was exercised *branch hits* number of times. The hits are bucketized to small powers of two. AFL refers this set of pairs as the *path* of an input. AFL says that an input achieves *new coverage* if it discovers a new (*branch ID*, *branch hits*) pair.

2.3 Limitations of AFL

While AFL's search strategy is guided by coverage, we observed in our experiments that often AFL fails to cover some important functionalities of the program under test. Note that achieving good coverage is precursor to finding bugs and crashes—if a program region is not covered, there is no way AFL can find bugs or crashes in that region.

Consider the code fragment shown in Figure 1. It is adapted from the `parser.c` file used in `libxml2`'s `xmllint` utility. AFL found many security vulnerabilities in this library in the past [51]. We ran AFL on this benchmark for 24 hours, repeating this experiment 20 times (see Section 4 for more experimental details). Only in one of these 24-hour runs did AFL produce an input passing Line 1. Even then, AFL failed to explore the contents of any of the if statements in Lines 6-30. As such, it failed to explore the large quantity of code after Line 31 (mostly omitted in Figure 1). Since this code is not even *covered*, then AFL simply cannot find any bugs in it.

The key reason AFL is unable to produce inputs covering any of this code—even after discovering an input containing `<!ATTLIST`—is that AFL mutates bytes paying no attention to which byte values are required to cover particular parts of the program. For example, after having produced the input `<!ATTLIST BD`, AFL will not prioritize mutation of the bytes after `<!ATTLIST`. Instead, it is as likely to produce the mutants `<!CATLIST BD`, `<!!ATTLIST BD`, or `???!ATTLIST BD` as it is to produce `<!ATTLIST ID`. However, to explore the code in Figure 1, once AFL discovers `<!ATTLIST BD`, it should not mutate the `<!ATTLIST` part of this input. To see why, suppose that the production of an input like `<!ATTLIST ID`—with the token "`ID`"—is required to pass the processing code omitted in Line 3 of Figure 1. Preventing the modification of `<!ATTLIST` increases AFL's probability of generating `<!ATTLIST ID` by at least 6×. Figure 2 illustrates how restricting mutation to only the last two characters of the input yields to a smaller space of mutants to explore, and thus, a higher probability of discovering an input that will get deeper into the program.

2.4 Overview of FairFuzz

We propose a two-pronged approach that addresses this concern but can be smoothly integrated into AFL or other mutation-based fuzzers. It works as follows.

The first part of our approach is the identification of statements like the if statement in Line 1 of Figure 1, which potentially guard large unvisited regions of code. For this, we utilize the observation that such statements are usually hit by very few of AFL's generated inputs (i.e. they are *rare*), and can thus be easily identified by keeping the track of the number of inputs which hit each branch. *Intuitively, the code guarded by a branch hit by few inputs is much less likely to have been thoroughly explored than the code guarded by a branch hit by a huge percentage of generated inputs.*

Having identified these rare branches for targeted fuzzing, we modify the input mutation strategy in order to keep the condition of the rare branch satisfied. Specifically, we use a deterministic mutation phase to approximately determine the parts of the input that cannot be mutated for mutants to hit the rare branch. The subsequent mutation stages are then not allowed to mutate these crucial parts of the input. As a result, we significantly increase the

```
if (CMP9(ptr,'<','!','A','T','T','L','I','S','T')) {
  ptr += 9;
  /* some processing code omitted */
  while ((ptr != '>') && (ptr != EOF)){
   int type = 0;
   if (CMP5(ptr,'C', 'D', 'A', 'T', 'A')){
       ptr += 5;
       type = XML_ATTRIBUTE_CDATA;
   } else if (CMP6(ptr,'I', 'D', 'R', 'E', 'F', 'S')){
       ptr += 6;
       type = XML_ATTRIBUTE_IDREFS\;
   } else if (CMP5(ptr,'I', 'D', 'R', 'E', 'F')){
       ptr += 5;
       type = XML_ATTRIBUTE_IDREF;
   } else if ((ptr == 'I') && ((ptr+1)== 'D')){
       ptr += 2;
       type = XML_ATTRIBUTE_ID;
   } else if (CMP6(ptr,'E','N','T','I','T','Y')){
       ptr += 6;
       type = XML_ATTRIBUTE_ENTITY;
   } else if (CMP8(ptr,'E','N','T','I','T','I','E','S')){
       ptr += 8;
       type = XML_ATTRIBUTE_ENTITIES;
   } else if (CMP8(ptr,'N','M','T','O','K','E','N','S')){
       ptr += 8;
       type = XML_ATTRIBUTE_NMTOKENS;
   } else if (CMP7(ptr,'N','M','T','O','K','E','N')){
       ptr += 7;
       type = XML_ATTRIBUTE_NMTOKEN;
   }
   if (type == 0) {ptr++; break;}

   /* more omitted code */

   if (CMP9(ptr,'#','R','E','Q','U','I','R','E','D')) {
     ptr += 9;
     default_decl = XML_ATTRIBUTE_REQUIRED;
   }
   if (CMP8(ptr,'#','I','M','P','L','I','E','D')) {
     ptr += 8;
     default_decl = XML_ATTRIBUTE_IMPLIED;
   }
   if (CMP6(ptr,'#','F','I','X','E','D')) {
     ptr += 6;
     default_decl = XML_ATTRIBUTE_FIXED;
     if (!IS_BLANK_CH(ptr)) {
       xmlFatalErrorMsg("Space required after '#FIXED'");
     }
   }
    ptr++;
  }
}
```

Figure 1: Code fragment based off the `libxml` file `parser.c` showing many nested if statements that must be satisfied to explore erroneous behavior.

probability of generating new inputs that hit the rare branch. This opens up the possibility of better exploring the part of the code that is guarded by the branch. While we apply it to targeting rare branches, this mutation modification strategy is general and can be applied to other testing targets.

We implement this approach on top of AFL in FairFuzz. We find this approach leads to faster coverage, as well as increased coverage compared to the maximum coverage achieved, over stock AFL and other modified versions of AFL on several real-world benchmarks. The details of the evaluation are presented in Section 4. We present the details of our approach in the next section.

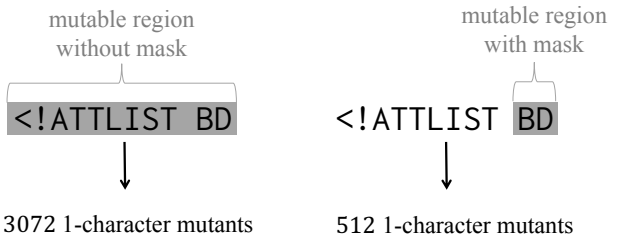

Figure 2: Preventing AFL from mutating the `<!ATTLIST` part of this input increases the probability of generating `<!ATTLIST ID` by at least 6×.

3 FAIRFUZZ ALGORITHM

In FairFuzz, we modify the AFL algorithm in two key ways to increase the coverage achieved. First, we modify the selection of inputs to mutate from the queue, and second, we modify the way mutations are performed on these inputs. Algorithm 2 outlines the FairFuzz algorithm and how it differs from the AFL algorithm. We begin with an abstract treatment of the mutation masking technique, and then dive into the FairFuzz particulars.

3.1 Mutation Masking

In this section we introduce the mutation mask for a given input, x, and a given testing target, T. We say $satisfies(x, T)$ is true if input x satisfies T.

Definition 1. A *mutation* is a tuple (c, m), where m is the number of bytes impacted by the mutation and c is one of the following mutation categories:

O: *overwrites* m bytes starting at position k with some values,
I: *inserts* some sequence of m bytes at position k,
D: *deletes* m bytes starting at position k.

To fully specify mutations with $c \in \{O, I\}$, the values that are inserted or written over existing bytes must be specified. Given an input x, a mutation $\mu = (c, m)$, and a position $i \in [0, |x| - m)$, let $mutate(x, \mu, i)$ denote the input produced by applying mutation μ on x at position i.

Definition 2. The *mutation mask* for an input x and a testing target T is a function $mask_{x,T} : \mathcal{N} \to \mathcal{P}(\{O, I, D\})$ which takes a position i in the input x and returns a subset of $\{O, I, D\}$. We say that a mutation category $c \in mask_{x,T}(i)$ if $satisfies(mutate(x, (c, 1), i), T)$ is true. That is, if c is in the set $mask_{x,T}(i)$, then after applying a mutation of category c at position i on x, the resulting input will satisfy the target T.

Intuitively, the mutation mask specifies whether the input produced from mutating x at position i will (likely) reach the testing target. With this mask, given a mutation $\mu = (c, m)$ at position k, we can compute

$$\textsc{okToMutate}(mask_{x,T}, \mu, k) = \bigwedge_{i=k}^{k+m-1} c \in mask_{x,T}(i).$$

We describe the algorithm to compute $mask_{x,T}(i)$ in Section 3.2.2.

Algorithm 2 FairFuzz algorithm. Differences from the AFL algorithm are highlighted in gray.

```
1:  procedure FairFuzz(Prog,Seeds)
2:    Queue ← Seeds
3:    while true do                                  ▷ begin a queue cycle
4:      for input in Queue do
5:        rarestBranch ← rarestHitBy(Prog, input,numHits)
6:        if numHits[rarestBranch] > rarity_cutoff then
7:          continue
8:        score ← performanceScore(Prog, input)
9:        mask ← computeMask(Prog, input, rarestBranch)
10:       for 0 ≤ i < |input| do
11:         for mutation in deterministicMutationTypes do
12:           if ¬okToMutate(mask, mutation, i) then
13:             continue
14:           newinput ←mutate(input, mutation, i)
15:           runAndSave(Prog,newinput, Queue)
16:       for 0 ≤ i < score do
17:         newinput ←mutateHavoc(input)
18:         runAndSave(Prog, newinput, Queue)
19: procedure mutateHavoc(Prog, input)
20:   numMutations ← randomBetween(1,256)
21:   newinput← input
22:   for 0 ≤ i <numMutations do
23:     mutation ← randomMutationType
24:     position ← randomOkToMutate(mask, mutation)
25:     newinput ← mutate(newinput, mutation, position)
26:   return newinput
27: procedure runAndSave(Prog, input, Queue)
28:   runResults ← run(Prog, input)
29:   for branch in runResults do
30:     numHits[branch] += 1
31:     if newCoverage(runResults) then
32:       addToQueue(input, Queue)
```

3.1.1 Biasing Mutation with the Mutation Mask. FairFuzz uses okToMutate($mask_{x,T}, \mu, k$) to bias mutations towards the testing target as follows.

In the deterministic mutation stages (Algorithm 2, Line 10-15), okToMutate is used to filter out mutants that could violate the testing target, as illustrated in Line 12 of Algorithm 2. In particular, for a given mutation type μ and position i, if okToMutate($mask_{x,T}, \mu, i$) is true, the mutant $x' = mutate(x, \mu, i)$ is generated and passed to runAndSave. Otherwise FairFuzz skips the mutant.

Recall that during the havoc stage, mutants are created by choosing a random mutation and random position at which to apply it. FairFuzz selects the random mutation $\mu = (c, m)$ as AFL does (Algorithm 2, Line 23). However, instead of selecting the position at random between 0 and $|newinput| - m - 1$, as in Algorithm 1, Line 20, FairFuzz chooses the position randomly from the subset of ok-to-mutate positions (Algorithm 2, Line 24). Precisely, the call to randomOkToMutate($mask_{x,T}, \mu$) in Line 24 of Algorithm 2 is:

$$sampleUniform(\{i \in [0, |x|-m-1] : \text{okToMutate}(mask_{x,T}, \mu, i)\}).$$

If the set of ok-to-mutate positions is empty, FairFuzz skips the mutation in Line 25 and chooses a new $\mu = (c, m)$ at the next iteration of the havoc mutation inner loop (Line 22).

3.2 Targeting Rare Branches

So far we have kept the testing target abstract. In this section, we concretize it by elaborating the definition of *rare branches* and giving the concrete algorithm which FairFuzz uses to compute the mutation mask for rare branches.

3.2.1 Selecting Inputs to Mutate. To bias input generation towards rare branches, FairFuzz selects only inputs that hit rare branches for mutation. First, we formalize the concept of a rare branch.

Definition 3. We say that an input x *hits* a branch b, denoted $hits(x, b)$, if the execution of the program on x exercises the branch b at least once.

The *hit count* of a branch is the number of produced inputs i which have exercised the branch. More formally,

Definition 4. Let $\mathcal{I}$ be the set of all inputs produced by fuzzing so far. The *hit count* of branch b is

$$numHits[b] = |\{x \in \mathcal{I} : hits(x, b)\}|.$$

Concretely, *numHits* is kept as a map of branches to hit count, updated every time a mutant is run (Line 30 of Algorithm 2). To establish *numHits*, FairFuzz runs one round of mutation on the seed input with no masking.

A natural idea is to designate the n branches hit by the fewest inputs as rare, or the branches hit by less than p percent of inputs to be rare. After some initial experiments, we rejected these methods as (a) they can fail to capture what it means to be rare (e.g. if $n = 5$ and the two rarest branches are hit by 20 and 15,000 inputs, both would be "rare"), and (b) these thresholds need to be modified for different benchmarks. Instead, we define a rare branch as one whose hit count is smaller than a dynamic rarity cutoff as follows. Let B be the set of all branches in the program.

Definition 5. Let $B_v = \{b \in B : numHits[b] > 0\}$. A *rare branch* is a branch b such that

$$numHits[b] \leq rarity\_cutoff$$

where

$$rarity\_cutoff = 2^i \text{ such that } 2^{i-1} < \min_{b' \in B_v} (numHits[b']) \leq 2^i.$$

For example, if the branch hit by the fewest inputs has been hit by 17 inputs, any branch hit by $\leq 2^5$ inputs is rare.

To determine whether an inputs hits a rare branch, FairFuzz computes the *rarest branch* hit by the input:

Definition 6. Let $branches(x) = \{b \in B : hits(x, b)\}$. Then the *rarest branch* hit by input x is the branch b^* such that

$$b^* = \operatorname*{arg\,min}_{b \in branches(x)} numHits[b].$$

Then, FairFuzz selects only inputs whose rarest branch is a rare branch for mutation (Line 6 of Algorithm 2).

Although FairFuzz only selects inputs using the above strategy, it could run some of the cycles using AFL's default strategy. This would ensure that it does not skip the default strategies of AFL, which might be better for creating crash-prone inputs.

Algorithm 3 Computing the mutation mask in FairFuzz.

```
1:  procedure computeMask(Prog, input, branch)
2:      mask ← initWithEmptySet(|input|)
3:      for 0 ≤ i < |input| do
4:          inputO ← mutate(input, flipByte, i)
5:          if branch ∈ branchesHitBy(Prog, inputO) then
6:              mask[i] ← mask[i] ∪ {O}
7:          inputI ← mutate(input, addRandomByte, i)
8:          if branch ∈ branchesHitBy(Prog, inputI) then
9:              mask[i] ← mask[i] ∪ {I}
10:         inputD ← mutate(input, deleteByte, i)
11:         if branch ∈ branchesHitBy(Prog, inputD) then
12:             mask[i] ← mask[i] ∪ {D}
        return mask
```

3.2.2 Computation of the Mutation Mask. Algorithm 3 outlines how FairFuzz computes $mask_{x,b}$ for a given input x and rare branch b. The algorithm works as follows.

For each position i in the x, FairFuzz produces the mutants x^O by flipping the byte at position i (Line 4 of Algorithm 3), x^I by adding a random byte at position i (Line 7), and x^D by deleting the byte at position i (Line 10). Then, for each x^c, FairFuzz determines whether $hits(x^c, b)$ by running x^c through the program (captured in branchesHitBy on Lines 5, 8, 11). Finally, if x^c hits b, FairFuzz notes the position i as overwritable (O), insertable (I), or deletable (D), respectively (Lines 6, 9, 12). While the calculation is illustrated as separate from the deterministic mutation stages in Algorithm 2, the two are integrated in the implementation. Since the mask computation adds only two new deterministic mutation types to AFL (byte-flipping is a default mutation type), the computation adds negligible overhead to stock AFL.

Of course, this computation of $O \in mask_{x,b}(i)$ and $I \in mask_{x,b}(i)$ is approximate—FairFuzz doesn't check whether every value overwritten or inserted results in b being hit. Unfortunately, trying all possible values to insert or write is too expensive and produces too many redundant inputs. Empirically we find this approximation produces an effective mutation mask (see Section 4.2).

Finally, note that this algorithm could be easily adapted to other testing targets by replacing $hits(x^c, b)$ with $satisfies(x^c, T)$.

3.3 Trimming Inputs for Testing Targets

AFL's efficiency depends on large part on its ability to quickly produce and modify inputs [54]. Thus, it is important to make sure the deterministic mutation stage—and in FairFuzz, mutation mask computation—is efficient. Since the runtime of the computation is linear in the length of the selected input, FairFuzz needs to keep the length of the inputs in the queue short. AFL has two techniques for keeping inputs short: (1) prioritizing short inputs when selecting inputs for mutation and (2) trimming (an efficient approximation of delta-debugging [55]) the parent input before mutating it. This trimming is omitted from Algorithms 1 and 2 for clarity. Trimming attempts to minimize the input to mutate with the constraint that the minimized input hits the same path (set of (*branch ID, branch hits*)) as the un-minimized one. However, this constraint is not good enough for reducing the length of inputs when very long inputs are chosen. FairFuzz may do this since it selects inputs based only on whether they hit a rare branch. We found that we can make inputs shorter in spite of this if we relax the trimming constraint. In particular, we relax the constraint to require that the minimized input hits only the target branch of the original input, instead of the same path as the original input. Similar relaxation could be done for other testing targets. We refer to FairFuzz with this relaxed constraint as FairFuzz *with trimming*.

4 IMPLEMENTATION AND EVALUATION

We implemented FairFuzz as an open source tool built on top of AFL. The implementation adds around 600 lines of C code to the file containing AFL's core implementation.

We evaluated FairFuzz on 9 different real-world benchmarks. We selected these from those favored for evaluation by the AFL creator (`djpeg` from libjpeg-turbo-1.5.1, and `readpng` from libpng-1.6.29), those used in AFLFast's evaluation (`tcpdump -nr` from tcpdump-4.9.0; and `nm`, `objdump -d`, `readelf -a`, and `c++filt` from GNU binutils-2.28) and a few benchmarks with more complex input grammars in which AFL has previously found vulnerabilities (`mutool draw` from mupdf-1.9, and `xmllint` from libxml2-2.94). Since some of these input formats had AFL dictionaries and some did not, we ran all the evaluation without dictionaries to level out the playing field. In each case we seeded the fuzzing run with the inputs in the corresponding AFL `testcases` directories (except `c++filt`, which was seeded with the input "`_Z1fv\n`"); for PNG we used only `not_kitty.png`.

4.1 Coverage Compared to Prior Techniques

In this section of evaluation we compare three popular versions of AFL against FairFuzz, all based off of AFL version 2.40b.

(1) "AFL" is the vanilla AFL available from AFL's website.
(2) "FidgetyAFL" [52] is AFL run without deterministic mutations.
(3) "AFLFast.new" [9] is AFLFast run without deterministic stage and with the cut-off-exponential exploration strategy.

Configurations (2) and (3) are the fastest-performing versions of AFL and AFLFast, respectively. We compare to 1) for baseline reference. We ran FairFuzz with input trimming for the testing target and omitting all deterministic stages except those necessary to compute the mutation mask.

We ran each technique for 24 hours (on a single core) on each benchmark. We repeated each 24 hour experiment 20 times for each benchmark. We ran our experiments for 24 hours as the fuzzing process does not have a defined end-time and this is a runtime used in prior work [11, 37, 42, 46]. We repeated our experiments 20 times because fuzz testing is an inherently non-deterministic process, and so is its performance. This enabled us to report results that are statistically significant in Section 4.1.1.

4.1.1 Overall Branch Coverage Achieved. We begin by analyzing coverage achieved by different techniques through time. The main metric we report is basic block transitions covered, which is close to the notion of branch coverage used in real-world software testing.

Why branch coverage? Other than basic block transitions (i.e., branches) covered, the only other commonly used metric to evaluate AFL coverage is *AFL path coverage*. As mentioned in Section 2.2,

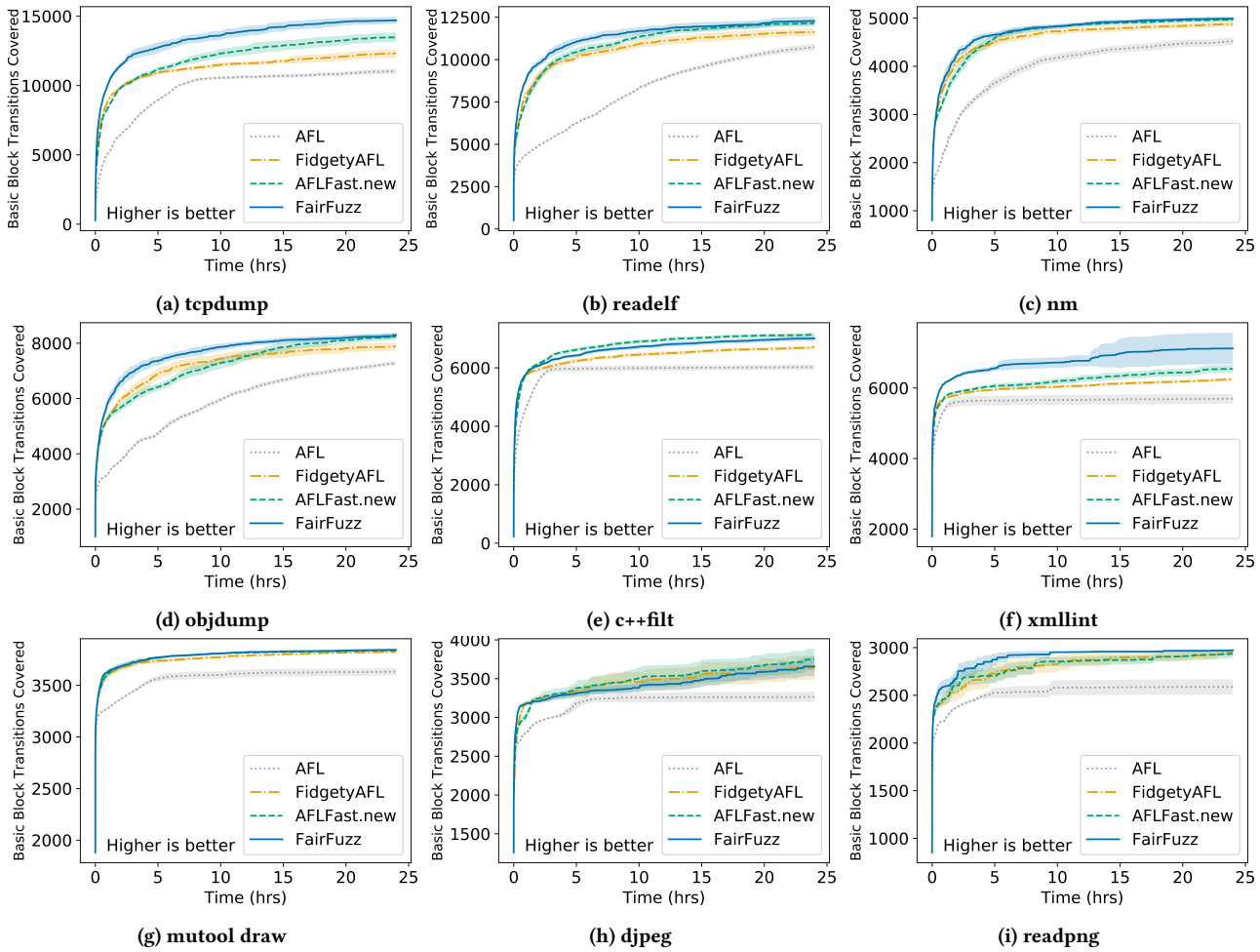

Figure 3: Number of basic block transitions (AFL branches) covered by different AFL techniques averaged over 20 runs (bands represent 95% C.I.s).

an AFL path is set of (*branch ID*, *branch hits*) But due to AFL's implementation, only the branches covered metric is robust to the order in which inputs are discovered. Here is a simple illustration of why the AFL path coverage is order-dependent. Consider a program with two branches, b_1 and b_2. Suppose input A hits b_1 once, input B hits b_2 once, and input C hits both b_1 and b_2. Their respective paths are $p_A = \{(b_1, 1)\}$, $p_B = \{(b_2, 1)\}$, and $p_C = \{(b_1, 1), (b_2, 1)\}$. If AFL discovers these inputs in the order A, B, C it will save both A and B and count 2 paths, and not save C since it does not exercise a new (*branch ID*, *branch hits*) pair. On the other hand, if AFL discovers the inputs in the order C, A, B, it will save C and count 1 path, and save neither A nor B. Thus it appears on the second run that AFL has found half the paths it did on the first run. On the other hand, regardless of the order in which inputs A, B, C are discovered, the number of branches covered will be 2. Reporting real path coverage is only possible if all inputs produced by AFL were saved, but this was not tractable in our 24-hour experiments given the volume of inputs produced (tens of millions). Thus, we report branch coverage. We also noted that the creator of AFL also favors branch coverage (which he refers to as "tuple resolution") as a performance metric, stating he has found it is the best predictor of how AFL will perform "in the wild" [53].

Results. Figure 3 plots, for each benchmark and technique, the average number of branches covered over all 20 runs at each time point (dark central line) and 95% confidence intervals in branches covered at each time point (shaded region around line) over the 20 runs for each benchmark. For the confidence intervals we assume Student's t distribution (taking 2.0860 times the standard error).

From Figure 3, we see that that on all benchmarks except `c++filt`, FairFuzz achieves the upper bound in branch coverage, generally showing the most rapid increase in coverage at the beginning of execution.

Note that while FairFuzz keeps a sizeable lead on the `xmllint` benchmark (Figure 3f), it does so with wide variability. Closer analysis reveals that one run of FairFuzz on `xmllint` was buggy, and no inputs were selected for mutation—this run covered no more than 6160 branches. However, FairFuzz had two runs on `xmllint` covering an exceptional 7969 and 10990 branches, respectively.

Figure 4, shows, at every hour, for how many benchmarks each technique has the *lead* in coverage. By *lead* we mean its average

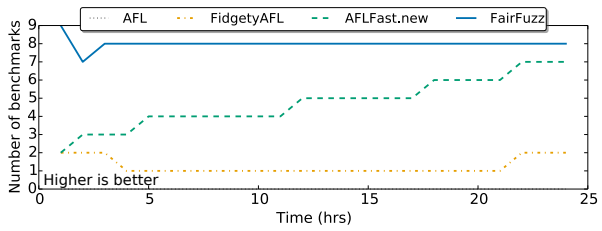

Figure 4: Number of benchmarks on which each technique has the lead in coverage at each hour. A benchmark is counted for multiple techniques if two techniques are tied for the lead.

coverage is above the confidence intervals of the other techniques, and no other technique's average lies within its confidence interval. We say two techniques are tied if one's average lies within the confidence interval of the other. If techniques tie for the lead, the benchmark is counted for both techniques in Figure 4, which is why the number of benchmarks at each hour may add up to more than 9. This figure shows that FairFuzz *quickly achieves a lead in coverage* on nearly all benchmarks and *is not surpassed* in coverage by the other techniques in our time limits.

4.1.2 Detailed Analysis of Coverage Differences. Figure 3 shows there are three benchmarks (`c++filt`, `tcpdump`, and `xmllint`) on which one technique achieves a statistically significant lead in AFL's branch coverage after 24 hours (with AFLFast.new leading on `c++filt` and FairFuzz on the other two). We were curious as to what these branch coverage increases corresponded to in terms of source code coverage differences.

Since AFL saves all inputs that achieve new program coverage (i.e. that are placed in the queue) to disk, we can replicate what program coverage was achieved in each run by replaying these queue elements through the programs under test. Since each benchmark was run 20 times, we take the union (over each technique) of inputs in the queue for all 20 runs. We ran the union of the inputs for each technique through their corresponding programs and then ran `lcov` on the results to reveal coverage differences. Using the union is a generous approach and can only reveal which regions are uncoverable by the different techniques over all the 20 runs.

`xmllint`. The bulk of the coverage gains on `xmllint` were in the main `parser.c` file. The key trend in increased coverage appears to be FairFuzz's increased ability to discover keywords.

For example, both AFL and FairFuzz have higher source code coverage than FidgetyAFL and AFLFast.new as they discovered the patterns `<!DOCTYPE` and `<!ATTLIST` in at least one run. However, FairFuzz also produced inputs satisfying *all the other conditionals* illustrated in Figure 1, which meant discovering all the keywords used in the comparisons. The produced inputs included:

```
<!DOCTYPET@[ <!ATTLIST?D T NMTOKENS
  <!DOCTYPE?[ <!ATTLIST D T ENTITY
<!DOCTYPE\[ <!ATTLISTíD T ID #REQUIRED^@^P
```

We believe the mutation masking technique is directly responsible for the discovery of these. To see this, let us focus on the

Table 1: Number of runs, for each technique, producing an input with the given sequence in 24 hours.

sequence	AFL	FidgetyAFL	AFLFast.new	FairFuzz
`<!A`	7	15	18	17
`<!AT`	1	2	3	11
`<!ATT`	1	0	0	1

`<!ATTLIST` block covered by the inputs above, whose code is outlined in in Figure 1. While both AFL and FairFuzz had a run discovering the sequence `<!ATTLIST`, of all the saved inputs for AFL in that run, only 0.4% of them (18) visited Line 2 of Figure 1, resulting in 18 hits of the line. In contrast, we found that 12.3% (1169) of the saved inputs produced by FairFuzz in the run where it discovered `<!ATTLIST` visited Line 2 of Figure 1, resulting *2124* hits of the line. With two orders of magnitude more hits of this line, it is obvious that FairFuzz was better able to explore the code in Figure 1. We believe the orders of magnitude difference can be attributed to the mutation mask.

To confirm the effect was not just luck, we also look at the number of runs which produced subsequences of `<!ATTLIST`. This is illustrated in Table 1. The decrease in the number of runs discovering `<!AT` from the number of runs discovering `<!A` in this table shows the mutation mask in action, with 11 of FairFuzz' runs discovering `<!AT`, compared to 1, 2, and 3 for AFLFast.new, FidgetyAFL, and AFLFast.new, respectively.

Finally, as is obvious from the example inputs above, although FairFuzz discovered more keywords, the inputs it produced were not necessarily more well-formed. Nonetheless, these inputs allowed the FairFuzz to explore more of the program's faults. This is reflected in the coverage of a large case statement differentiating 57 error messages in `parser.c`. These messages do not result in AFL "crashes" (i.e. segmentation faults), simply in non-zero exit codes. Both FidgetyAFL and AFLFast.new cover only 22 of these cases, AFL covers 33, and FairFuzz covers 39.

`tcpdump`. Like `xmllint`, `tcpdump` has extensive nested structure, with the presence of various sequences in the input leading to different printing functions. We observed that coverage for `tcpdump` differs a bit for all four techniques over a variety of different files, but see the biggest gains in three files printing certain packet types (`print-forces.c`, `print-llc.c`, and `print-snmp.c`).

The coverage gains in these files suggest FairFuzz is better able to automatically detect sequences in the inputs necessary to increase program coverage. For example, unlike the other three techniques, FairFuzz was able to create files that have legal ForCES (RFC 5810) packet length. FairFuzz was also able to create IEEE 802.2 Logical Link Control (LLC) packets with the organizationally unique identifier (OUI) corresponding to RFC 2684, and subsequently explore the many subtypes of this OUI. Finally, in the Simple Network Management Protocol parser, FairFuzz was able to create inputs corresponding to Trap PDUs, as well as some inputs with a correct SNMPv3 user-based security message header.

We note these gains in coverage seem less impressive than those of FairFuzz on `xmllint`, even though the performance in Figure 3 looks similar. This appears to be because FairFuzz gets consistently higher coverage of `tcpdump` instead of covering parts of

the program wholly uncoverable by the other techniques. We can see this by looking at the number of branches covered *by at least one of* the 20 runs (the union over the runs) and the number of branches covered *at least once in all* the 20 runs (the intersection over the runs). For `tcpdump`, FairFuzz has a consistent increase in the intersection of coverage (FairFuzz's contains 11,293 branches compared to AFLFast.new's 10,724), but a smaller gain in the union (FairFuzz's is 16,129, while AFLFast.new's is 15,929). On the other hand, the intersection of coverage for `xmllint` is virtually the same for all techniques except stock AFL (5,876 for FidgetyAFL, 5,778 for AFLFast.new and 5,884 for FairFuzz, maybe because of the buggy run mentioned in Section 4.1.1), but FairFuzz's union of coverage (11,681) contains over 4,000 more branches than that of AFLFast.new (7,222).

`c++filt`. The differences in terms of source code coverage between techniques were much more minimal for `c++filt` than for `tcpdump` or `xmllint`. For example, FairFuzz covers 3 lines in `cp-demangle.c` that AFLFast.new does not, related to demangling binary components when the operator has a certain opcode. On the other hand, AFLFast.new covers a branch where `xmalloc_failed(INT_MAX)` is called when a length bound comparison fails, while FairFuzz fails to produce an input long enough to violate the length bound. FairFuzz also fails to cover a branch in `cxxfilt.c` taken when the length of input read into `c++filt` surpasses the length of the input buffer allocated to store it.

FairFuzz's inability to produce very long inputs may be related to the second round of trimming FairFuzz does. Or, it could be because `c++filt` has highly recursive structure, so full branch coverage is not as good a exploration heuristic for this program. A testing target other than hitting rare branches may be better suited for programs like `c++filt`.

The pattern we see from this analysis is that FairFuzz is better able to automatically discover input constraints and keywords—special sequences, packet lengths, organization codes—and target exploration to inputs which satisfy these constraints than the other techniques. We suspect the gains in coverage speed on benchmarks such as `objdump`, `readpng`, and `readelf` are due to similar factors. We conjecture the targeting of rare branches shines the most in the `tcpdump` and `xmllint` benchmarks since these programs are structured with many nested constraints, which the other techniques are unable to properly explore over the time budget (and perhaps even longer) without extreme luck.

4.2 Can Masking Effectively Target Branches?

Finally, we were curious as to whether the mutation mask strategy effectively biased mutation towards our testing target. In FairFuzz, the target was hitting the same rare branch as the parent input. We conducted the following experiment on a subset of our benchmarks to evaluate the effect of the mask.

We added a *shadow mode* to FairFuzz. When running in shadow mode, every time an input is selected for mutation, FairFuzz first performs all mutations without the influence of the mutation mask (the *shadow* run). Then, for the same input, FairFuzz performs all mutations again, using the mutation mask filtering and bias.

This shadow run allows us to compute the difference between the percentage of generated inputs hitting the target with and without the mutation mask *for each parent input*. Since some target branches may be easier to hit than others, this gives us a better idea of how effective the masking technique is in general. In our experiments, we ran FairFuzz with the shadow run on a subset of our benchmarks. For each benchmark we ran a cycle with target branch trimming and one without.

Table 2: Average % of mutated inputs hitting target branch for one queueing cycle.

(a) Cycle without trimming.

	det. mask	det. plain	hav. mask	hav. plain
`xmllint`	92.8%	46.5%	31.8%	6.6%
`tcpdump`	99.0%	74.0%	34.2%	9.3%
`c++filt`	97.6%	64.1%	41.4%	14.4%
`readelf`	99.7%	82.7%	57.7%	14.9%
`readpng`	99.1%	34.6%	24.3%	2.4%
`objdump`	99.2%	70.2%	42.4%	9.0%

(b) Cycle with trimming.

	det. mask	det. plain	hav. mask	hav. plain
`xmllint`	90.3%	22.9%	32.8%	2.9%
`tcpdump`	98.7%	72.8%	36.1%	9.0%
`c++filt`	96.6%	14.8%	34.4%	1.1%
`readelf`	99.7%	78.2%	55.5%	11.4%
`readpng`	97.8%	39.0%	24.0%	2.4%
`objdump`	99.2%	66.7%	46.2%	7.6%

Our results are presented in Table 2, which shows the target branch hit percentages for the deterministic and havoc stages. These percentages are the averages—over all inputs selected for mutation in the first queueing cycle—of the percentage of children inputs hitting the target.

Overall, Table 2 shows that the mutation mask largely increases the percentage of mutated inputs hitting the target branch. The hit percentages for the deterministic stage are strikingly high. This is not unexpected because in the deterministic stage the mutation mask simply prevents mutations at locations likely to violate the target branch. Thus, the gain percentage of inputs hitting the target branch in the havoc stage is most impressive. In spite of the use of the mutation mask in the havoc stage being heuristic, we consistently see the use of the mutation mask causing a 3x-10x increase in the percentage of inputs hitting the target branch. As for trimming, it appears that extra trimming reduces the number of inputs hitting the target branch when the mutation mask is disabled but has minimal effect when the mutation mask is enabled.

Again, we note that the mutation masking technique is independent of the testing target. In particular, the fact that the branches being targeted are "rare". This suggests that this strategy could be used in a more general context. For example, we could target only the branches within a function that needs to be tested, or, if some area of the code was recently modified or bug-prone, we could target the branches in that area with the mutation mask. Recent work on targeted AFL [10] shows promise in such an application of AFL, and we believe the mutation masking technique could be used cooperatively with the power schedules presented in that work.

```
if (strcmp(str,"BAD!")) {
 // do bad things
}
```

```
if (str[0] == 'B') {
 if (str[1] == 'A') {
  if (str[2] == 'D') {
   if (str[3] == '!') {
    // do bad things
}}}}
```

Figure 5: Multi-byte comparison (left) unrolled to byte-by-byte comparison (right).

5 DISCUSSION

While we chose benchmarks with a variety of input formats accomplishing different tasks, the results of our evaluation may not generalize to other programs.

The foremost limitation of using rare branches as a testing target in FairFuzz is the fact that branches that are never hit by any AFL input cannot be targeted by this method. So, it confers little benefits to discovering a single long magic number when progress towards matching the magic number does not result in new coverage—e.g., the comparison on the left of Figure 5. We believe the FairFuzz mutation masking algorithm could be used in conjunction with methods targeting the magic number issue [37, 42, 46] to build a more effective fuzzer.

Recall FairFuzz was effective at finding keyword sequences in the `xmllint` benchmark. This may be because the long string comparisons in `parser.c` use `CMPn` macros (see Figure 1), which are structured as byte-by-byte comparisons. So, AFL's instrumentation reports new coverage when progress was made on these comparisons. The creators of `laf-intel` [1] propose several LLVM "deoptimization" passes to improve AFL's performance, including a pass that automatically turns multi-byte comparisons into byte-by-byte comparisons. Figure 5 shows an example of this comparison unrolling. The integration of these LLVM passes into AFL's instrumentation is straightforward, requiring only a patch to AFL's LLVM-based instrumenter [2]. Due to FairFuzz's performance on `xmllint`, we believe FairFuzz could show similar coverage gains on other programs if they were compiled with this `laf-intel` pass. We did not evaluate this as the evaluation of the `laf-intel` pass was done in AFL's "parallel" fuzzing mode. We did not do any experiments with this parallel fuzzing as our implementation did not have a distributed version of the rare branch computation algorithm.

6 OTHER RELATED WORK

We have discussed AFLFast [11] in the previous sections. Unlike our proposed approach, it targets rare paths, not branches, and it does not change the mutation strategies of AFL. Other prior work on AFL has focused on producing a single input passing a difficult to hit branch [37, 46], like the one on the left of Figure 5. Driller [46] uses symbolic execution to pass branches when AFL gets stuck. Steelix [37], whose source code was unavailable at the time of submission, adds a static analysis stage, extra instrumentation, and mutations to AFL to better produce inputs satisfying multi-byte comparisons. These techniques require more instrumentation but find magic numbers more accurately than FairFuzz. However, neither of these techniques will be able to prevent a discovered magic sequence from being mutated to encourage further exploration, while FairFuzz does.

Unlike FairFuzz and other greybox fuzzers [3] which use coverage information as a heuristic for which inputs may yield new coverage under mutation, symbolic execution tools [13, 22, 44, 45] methodically explore the program under test by capturing path constraints and directly producing inputs which fit yet-unexplored path constraints. The cost of this precision is that it can lead to the path explosion problem, which causes scalability issues.

Traditional blackbox fuzzers such as zzuf [30] mutate user-provided seed inputs according to a mutation ratio, which may need to be adjusted to the program under test. BFF [33] and SymFuzz [14] adapt this parameter automatically, by measuring crash density and doing input bit dependence, respectively. These optimizations are not relevant to AFL-type fuzzers which do not use this mutation ratio parameter.

There exist several fuzzers highly optimized for certain input file structures, including network protocols [4, 12], and source code [31, 43, 49]. FairFuzz is of much lower specificity so will not be as effective as these tools on these specific input formats. However, its method is fully automatic, requiring neither user inputs [4] or extensive tuning [43, 49].

While FairFuzz uses its mutation mask to try and fix important parts of program inputs, recent work has more explicitly tried to automatically learn input formats. Learn&Fuzz [23] uses sequence-based learning methods to learn the structure of PDF objects, Autogram [32] proposes a taint analysis-based approach to learning input grammars, while Glade [8] uses an iterative approach and repeated calls to an oracle to learn a context-free grammar for a set of inputs. FairFuzz does not assume a corpus of valid inputs of any size from which validity could be automatically learned, nor an oracle for the grammar, but consequently does not learn as precise a grammar of the input.

Another approach to smarter fuzzing is to find locations in seed inputs related to likely crash locations in the program and focus mutation there [20, 26, 48] and TaintScope [48]. These directed methods are not directly comparable to FairFuzz since they do not have the same goal of increasing program coverage. VUzzer [42] uses both static and dynamic analysis to get immediate values and input positions used in comparisons. It uses a Markov Chain model to decide which parts of the program should be targeted, as opposed to our empirical approach.

Randoop [41] automatically generates test cases for object oriented-programs through feedback-directed random test generation. Evosuite [19] uses seed inputs and genetic algorithms to achieve high code coverage. Both these techniques focus on generating sequences of method calls to test programs, not byte-sequence inputs for a given program like FairFuzz.

Search-based software testing (SBST) [24, 25, 27–29, 35, 38, 39, 50] uses optimization techniques such as hill climbing and genetic algorithms to generate inputs that optimize some observable fitness function. These techniques work well when the fitness curve is smooth with respect to changes in the input, which is not the case in coverage-based greybox fuzzing.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants CCF-1409872 and CCF-1423645. Thanks to Rohan Padhye, Kevin Laeufer, and the anonymous reviewers for their extensive feedback on this paper.

REFERENCES

[1] 2016. laf-intel. https://lafintel.wordpress.com/. Accessed August 23rd, 2017.
[2] 2016. laf-tintel source. https://gitlab.com/laf-intel/laf-llvm-pass. Accessed August 24th, 2017.
[3] 2016. libFuzzer. http://llvm.org/docs/LibFuzzer.html. Accessed August 25th, 2017.
[4] Pedram Amini and Aaron Portnoy. 2012. Sulley. https://github.com/OpenRCE/sulley. Accessed August 22nd, 2017.
[5] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2007. JPF-SE: a symbolic execution extension to Java PathFinder. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
[6] Abhishek Arya and Cris Neckar. 2012. Fuzzing for Security. https://blog.chromium.org/2012/04/fuzzing-for-security.html.
[7] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1083–1094.
[8] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*.
[9] Marcel Böhme. 2016. AFLFast.new. https://groups.google.com/d/msg/afl-users/1PmKJC-EKZ0/lbzRb8AuAAAJ. Accessed August 23rd, 2017.
[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.
[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*.
[12] Sergey Bratus, Axel Hansen, and Anna Shubina. 2008. *LZfuzz: a fast compression-based fuzzer for poorly documented protocols*. Technical Report. Department of Computer Science, Darmouth College.
[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*.
[14] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*.
[15] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*.
[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*. 30, 1 (2012), 2.
[17] Lori A. Clarke. 1976. A program testing system. In *Proc. of the 1976 annual conference*. 488–491.
[18] Chris Evans, Matt Moore, and Tavis Ormandy. 2011. Fuzzing at Scale. https://security.googleblog.com/2011/08/fuzzing-at-scale.html. Accessed August 24th, 2017.
[19] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*.
[20] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*.
[21] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*.
[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*.
[23] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. *CoRR* (2017). http://arxiv.org/abs/1701.07232
[24] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 156–166.
[25] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 408–418.
[26] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*.
[27] Mark Harman. 2007. The current state and future of search based software engineering. In *2007 Future of Software Engineering*. IEEE Computer Society, 342–357.
[28] Mark Harman and John Clark. 2004. Metrics are fitness functions too. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*. IEEE, 58–69.
[29] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.
[30] Sam Hocevar. 2007. zzuf. http://caca.zoy.org/wiki/zzuf/. Accessed August 22nd, 2017.
[31] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*.
[32] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*.
[33] Allen D. Householder and Jonathan M. Foote. 2012. *Probability-Based Parameter Selection for Black-Box Fuzz Testing*. Technical Report. Carnegie Mellon University Software Engineering Institute.
[34] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19 (July 1976), 385–394. Issue 7.
[35] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.
[36] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *CAV*. 609–615.
[37] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*.
[38] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*. IEEE Computer Society, Washington, DC, USA, 153–163. https://doi.org/10.1109/ICSTW.2011.100
[39] Webb Miller and David L. Spooner. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering* 2, 3 (1976), 223.
[40] Max Moroz and Kostya Serebryany. 2016. Guided in-process fuzzing of Chrome components. https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html.
[41] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*.
[42] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*.
[43] Jesse Ruderman. 2015. jsfunfuzz. https://github.com/MozillaSecurity/funfuzz/tree/master/js/jsfunfuzz.
[44] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*.
[45] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*.
[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*.
[47] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex - White Box Test Generation for .NET. In *Proceedings of Tests and Proofs*.
[48] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*.
[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.
[50] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 140–150.
[51] Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Accessed August 18th, 2017.
[52] Michał Zalewski. 2016. FidgetyAFL. https://groups.google.com/d/msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ. Accessed August 23rd, 2017.
[53] Michał Zalewski. 2016. Unique crashes as a metric. https://groups.google.com/d/msg/afl-users/fOPeb62FZUg/LYxgPYheDwAJ. Accessed August 24th, 2017.
[54] Michał Zalewski. 2017. American Fuzzy Lop Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed August 18th, 2017.
[55] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.